A Language for Bitmap Manipulation

LEO J. GUIBAS and JORGE STOLFI
Xerox PARC and Stanford University

This paper proposes that bitmaps, or raster images, should be given full citizenship in the world of computer science. We introduce a calculus of bitmap operations and MUMBLE, a programming language appropriate for describing bitmap computations. We illustrate the use of MUMBLE by several interesting graphical applications. We also discuss the structure of BOP, an efficient implementation of the bitmap calculus that is the underpinning of our system.

CR Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*very high-level languages*; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*display algorithms;* I.3.4 [**Computer Graphics**]: Graphics Utilities—*graphics packages*; *picture description languages*; *software support*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*languages*; I.4.0 [**Image Processing**]: General—*image processing software*; I.4.1 [**Image Processing**]: Digitization—*quantization*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Raster image, bitmap calculus, delayed evaluation, bitblt instruction

1. INTRODUCTION

Many present-day graphical systems use raster devices, such as displays and printers, as output media. These media require that displayed pictures and text first be converted into *bitmaps*, (i.e., arrays of discrete intensity/color values). Some such systems, notably the new breed of personal computers initiated by the Xerox Alto, the M.I.T. Lisp machine, and others, have been able to provide a user interface superior to anything that has existed before through the use of specifically designed, powerful bitmap manipulation instructions.

Up to now the study of such instructions has not received extensive treatment in the literature. It has been considered mostly the province of system architects and a few daring hackers. Furthermore, bitmaps have been treated as imperfect and mathematically uninteresting approximations to the ideal images of plane geometry. We feel that it is time to give bitmaps full citizenship in the world of

The work of Jorge Stolfi, who is on leave from the University of Sao Paulo (Sao Paulo, Brazil), was partially supported by a grant from Conselho Nacional de Desenvolvimento Cientifico e Tecnologico (CNPq).

Authors' present addresses: Dr. Leo J. Guibas, Computer Science Laboratory, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, Calif. 94304; Jorge Stolfi, Computer Science Department, Stanford University, Stanford, Calif. 94305.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0730-0301/82/0700-0191 $00.75

computer science. Their practical importance alone would be enough to justify this attention, but it can also be argued that bitmaps are worthy mathematical objects on their own.

In this paper we propose a *bitmap calculus*: a mathematical model and notation for describing operations on bitmaps. The model is machine-independent and takes full account of the discrete nature of bitmaps. It is quite general, allowing one to describe, for example, color and grey-scale computations, point and neighborhood operators, picture translation, rotation, and sampling. We show some obvious and not so obvious applications of this calculus. It is interesting that there are several examples of bitmap computations that ostensibly depend on global properties, and yet in fact can be done by applying purely local operations in a uniform manner.

We decribe MUMBLE, an interactive language designed and implemented to allow experimentation within this bitmap calculus. It is hoped that such experimentation will help in the discovery and development of useful algorithms and "idioms" of the bitmap calculus; we mention some interesting examples that seem to justify this hope.

Finally, we present some techniques that can be used to evaluate complex bitmap expressions efficiently. In particular, we show how to use "lazy evaluation," run-length encoding, and word-oriented Boolean operations to cut down on time and memory requirements. In the evaluation of complex bitmap expressions, for example, these techniques will avoid the creation of large temporary bitmaps. These ideas have been implemented in BOP, a flexible and general software package for bitmap manipulation, that forms the underpinning of our MUMBLE system.

Section 1 of this paper discusses a justification for bitmaps and introduces the *bitblt* instruction; Section 2 presents our bitmap calculus; Section 3 describes MUMBLE; Section 4 contains a collection of MUMBLE programs that do various interesting graphic computations; Section 5 elaborates on BOP; and finally Section 6 contains some conclusions.

1.1 The Traditional View of Bitmaps

Traditionally, bitmaps have been viewed as imperfect approximations to the ideal images one would like to display. Such ideal images are assumed to be described by continuous variation in color, intensity, and so forth. The imperfection is brought about by the discrete nature of the imaging devices we possess, as well as the digital nature of the computers themselves, which forces us to quantize these continuous variables into a discrete (but possibly very large) set of values. A consequence of this view is that bitmaps are essentially an implementation artifact, and the graphics programmer should be spared the pain of being exposed to them as long as possible. For example, in designing a graphics package it has often been argued that the objects the programmer should be given access to are the ideal shapes of plane geometry plus continuous functions for describing color and intensity modulation. The operators available in such a package must always correspond to operations on these ideal objects. It is the responsibility of the package's implementor to discretize all these continuous functions internally and transform the ideal continuous operators into their discrete counterparts.

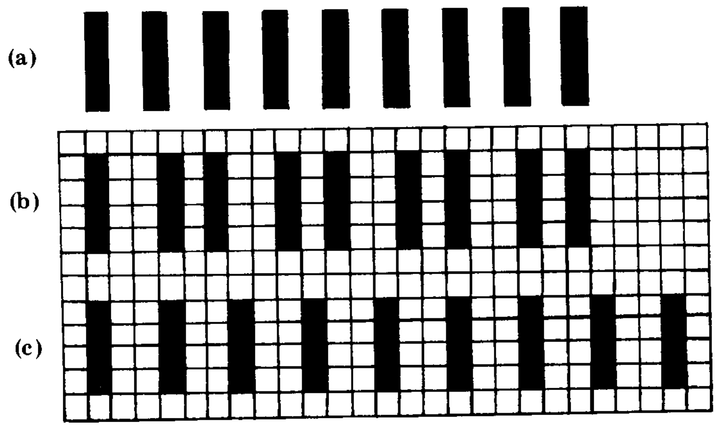

Fig. 1. A picket fence image (a) and two discretized versions of it: (b) globally faithful, and (c) locally faithful.

Unfortunately, for a variety of reasons, no such package has ever been able to hide fully the discrete representation involved beneath the surface. Perhaps the most fundamental reason is that certain laws, which hold in the continuous domain, cannot be made to hold in the discrete domain, no matter how careful one is about the quantization process. (We will have more to say about this shortly.) Also, good quantized approximations to continuous tone images naturally involve very large bitmaps. Such large bitmaps are expensive to store and manipulate. Thus efficiency considerations often make it imperative that the graphics programmer be made aware of the implementation underneath.

As anyone who has taken a course in numerical methods knows, floating-point numbers do not satisfy many of the identities that hold true for real (ideal) numbers. In large part, the science of numerical computing is devoted to compensating for these (fundamental) imperfections in floating-point arithmetic. Similarly, when images are discretized, certain errors are unavoidable. In general, different policies about how these errors are to be made can optimize different goals. Two philosophies commonly used for error distribution are what we shall term *global faithfulness* and *local faithfulness*. These are illustrated by considering the picket fence example shown in Figure 1. Suppose each picket is 1 pixel wide, but the pickets are spaced 2.5 pixels apart. Then the global faithfulness method would rasterize the picket spacing to an alternating thickness of 2 and 3 pixels respectively, so as to make the overall length of the fence as close to its real value as possible. The local faithfulness method would simply choose either 2 or 3 pixels and make all spacings that thickness. In the global method the overall length of the fence is preserved, but the local congruence of the spacings is lost. In the local method the converse is true: local congruence is preserved, but global dimensions can be way off. Which scheme is better clearly depends on the particular application.

Another reason bitmaps seep through to higher levels of system design is efficiency. Dynamic raster graphics require very high-speed manipulations of the raster memory [17], where the image being displayed is stored. In every case, the contents of the raster memory can be computed by scan conversion algorithms from higher level shape, illumination, and color descriptions. Such algorithms are rarely, however, fast enough to cope with incremental updating of the display, as is often required in interactive applications. Although, as we remarked earlier, ideal image manipulations do not always have exact counterparts in discrete form, in many cases they often do. For instance, the common operation of copying or translating a subimage on the screen has an exact counterpart in the discretized form. There are immense speed advantages to be gained by doing these manipulations in the raster representations, rather than in the ideal ones and then repeating the scan conversion.

It is the conclusion of this subsection that an attempt to ban bitmaps from anything other than a temporary low-level representation for computer images is bound to encounter difficulties.

1.2 Introduction of Bitblt

To facilitate raster manipulations, several novel computer systems have used specialized instructions dealing with the raster memory. Typically these systems have been personal computers, such as the Xerox Alto [18] or the M.I.T. Lisp Machine [21], where a premium is placed on interactive graphics facilities. Their raster instructions are powerful primitives, often implemented in a combination of hardware and microcode. For example, the operation of copying a rectangle of pixels from one part of the screen to another can be accomplished by invoking a single such instruction. Such an instruction is commonly known by the name *bitblt* (bit boundary block transfer), or *RasterOp* in the Newman–Sproull terminology [15]. (See also the discussion in [1].) The first instance of this instruction was coded by Dan Ingalls and Diana Merry at Xerox PARC in 1975.

The most common form of this instruction is a bitwise Boolean operation between two comformable (i.e., having the same dimensions) and possibly overlapping rectangles of pixels. If we call one rectangle S for *source* and the other D for *destination*, then bitblt performs the operation $d \leftarrow d \odot s$ for corresponding bits d and s in D and S. Here $\odot$ denotes some two-argument Boolean operation, which is a parameter to bitblt. Such an instruction can obviously be used to move rectangles of bits around the screen. There are many subtle issues regarding this instruction. How are the two rectangles to be specified? In whose coordinate system? What if they are not comformable? Useful stipple and grey-scale patterns can be obtained by giving an appropriate meaning to bitblt when mapping a small rectangle onto a large one. There are also several interesting issues about the implementation of the bitblt instruction, but we will not elaborate on these here.

For us, the most important property of bitblt is that it has been found to have an amazing number of uses, far beyond the obvious ones discussed above. (This fact has been part of the raster graphics folklore for some time.) For example, it is possible to rotate or transpose an $n \times n$ bitmap with a number of bitblts which is a small constant times n. Bitblt can also be used to fill in areas, count connected components, and do several other interesting and useful bitmap computations in very ingenious ways. Part of the reason for this seems to be that many interesting bitmap computations can be done entirely through local operations, that is by uniformly replacing each pixel with a function of (the old values of) itself and its neighboring pixels. Such algorithms have been studied to a certain extent in the theory of iterative arrays [20] (though not especially in a graphics context) and correspond naturally to invocations of bitblt. We will see some examples later in this paper.

2. A BITMAP CALCULUS

The techniques mentioned in the previous section appear at first to have an ad hoc nature, to be just hacks or tricks. We believe instead that they are instances of general mathematical principles waiting to be discovered, if an appropriate setting is created. Such a setting would be a calculus of bitmap operations, so one can learn to use these operations just as naturally as arithmetic operations on numbers.

A number of related calculi have already established their value. APL [9, 16] is a widely used programming language whose basic data structure is the multidimensional array. Much of the power of APL stems from the fact that it has a carefully chosen set of operators which compose well with each other. In fact APL was invented as a notation for array computations long before an interpreter for it was available. Many of the issues we will discuss concerning efficient implementation of bitmap computations have also been encountered earlier in the construction of optimized APL interpreters and compilers [6].

Boolean algebra is another calculus closely related to our bitmap computations, since very frequently our bitmaps are only a single bit deep. Although most of us have some experience with manipulating expressions involving ANDs and ORs, perhaps we do not all realize the usefulness of other Boolean functions, such as exclusive OR (XOR). The XOR operation also satisfies simple algebraic laws, for instance

$$(x \text{ XOR } y) \text{ XOR } x = y,$$

or

$$(x \text{ XOR } y) \text{ AND } z = (x \text{ AND } z) \text{ XOR } (y \text{ AND } z).$$

This can often be used to advantage. Some of the most interesting uses of XOR are covered in [10]. Several applications of XOR in bitmap computations are discussed later in this paper.

The most significant influences on the bitmap language we describe have been these two calculi, plus a hefty amount of intuition about the proper geometric shapes to use for creating primitive bitmaps (the atoms of this calculus).

The concepts and a notation for the bitmap calculus are defined in what follows. This description should be considered a proposal of a formal mathematical notation for bitmaps and bitmap operations, rather than the definition of a programming language. However, the syntax and semantics proposed here were also incorporated in MUMBLE, as FORTRAN and ALGOL were patterned after the notation of common algebra.

2.1. Pictures and the Image Plane

A *(discrete) image*, or *picture*, is defined to be an infinite two-dimensional array of bitstrings of uniform length called *pixels* (short for *picture elements*). Pictures are the central concept of the bitmap calculus and are a mathematical representation of the intuitive "bitmap" concept. The elements of a given picture all have the same length, which is called the *depth* of the picture. They are indexed by pairs $[i, j]$, where i and j range over all (positive and negative) integers. We follow the array convention when drawing the coordinate axes, so the i-axis proceeds downward and the j-axis to the right. This is just a 90 degree clockwise rotation of the normal x, y-coordinate system.

It is convenient to think of a picture B in geometric terms, by considering its pixels to be located on a Cartesian plane (the *image plane*), with pixel $\mathsf{B}_{[i,j]}$ centered at the point with coordinates $[i + \frac{1}{2}, j + \frac{1}{2}]$.[1] Also, the bits $b_{n-1} b_{n-2} \cdots b_1 b_0$ of each pixel are considered to be stacked in the direction perpendicular to the image plane, with b_0 at the bottom. We can therefore speak of the kth layer of the image as the picture formed by considering bit b_k of every pixel.

Operations and algorithms involving pictures make frequent use of points, vectors, and other geometric entities of the image plane, particularly those with integer coordinates. An important example is that of a *box*: a rectangular region of the plane with sides parallel to the axes. We represent a box by a pair of points $[l, h]$ with integer coordinates, the coordinates of the upper left and lower right corner, respectively. A pixel is said to be inside a box (or any other geometric figure) if the same is true of its center point.

2.2 Elementwise Operations

Most operations defined for finite bit strings naturally extend to pictures in an elementwise fashion. For example, pixels can be combined by using bitwise Boolean operations such as AND, OR, XOR, etc. Bitstrings can also be interpreted as representing numbers in binary notation, and thus the usual arithmetic and relational operations extend naturally to pictures. Signed (two's complement) and unsigned binary notations are both widely used in computer practice; each has its own advantages and a large set of applications for which it is almost mandatory. Both notations have been retained in the bitmap calculus.

Relational operators (=, >, etc.) compare the numerical values of the operands, and return either 0 (= FALSE) or -1(= TRUE) in two's complement. Relational operators extended to pictures are very useful in building *masks*: one-bit images containing only TRUE and FALSE pixels. Masks can be used with Boolean operators to extract and combine selected parts of other pictures. For example, (X AND (A > B)) OR (Y AND (A ≤ B)) is an image that is equal to $\mathsf{X}_{[i,j]}$ wherever $\mathsf{A}_{[i,j]} > \mathsf{B}_{[i,j]}$, and equal to $\mathsf{Y}_{[i,j]}$ everywhere else.

Other bitstring operations that are particularly useful when extended to pictures are: b TOLEN n, which makes b into an n-bitstring by padding or truncating its high-order bits; b CHOP n, which returns the higher order n bits of b; and the *lamination* operation, $a \mid b$, which concatenates the bitstrings a and b. In addition, the depth and the kth layer of a picture P are denoted by DEPTH P and P LAYER k, respectively.

Conditional bitmap expressions can be defined in terms of Boolean operators: IF A THEN B ELSE C FI is by convention equivalent to B AND (A ≠ FALSE) OR

[1] This "half-off" convention leads to cleaner definitions for rotations, boxes, etc.

C AND (A = FALSE), for any pictures A, B, and C. A related operation is P CUT b which produces a picture coinciding with P inside the box b, but *zero everywhere else.*

When two bitstrings of unequal length are combined in a Boolean operation, we assume that the shorter is extended (preserving its numerical value) to match the longer. This means that TRUE and FALSE extend to the bitstrings 111 . . . 1 and 000 . . . 0, and therefore masks built by relational operators can be used with pictures of any depth.

2.3. Geometric Operations

Let T be a *geometric transformation,* a function that maps the image plane into itself, and let P be a picture. The picture T(P) is defined by moving every pixel of P as specified by T. For example, consider the operation p SHIFT v, that displaces the point $p = [p_i, p_j]$ by the vector $v = [v_i, v_j]$, producing the point $[p_i + v_i, p_j + v_j]$. The result of P SHIFT [7, 3] is a picture Q such that $Q_{[i,j]} = P_{[i-7,j-3]}$ for all $[i, j]$. Geometric operations can be extended to boxes and other geometric figures in exactly the same way.

In order for this extension to have meaning, the function T must have a well-defined inverse, even though T itself may be undefined or multivalued at some points. The function T is assumed to apply to the *centers* of the pixels, rather than to their indices, so $T^{-1}(p)$ must map pixel centers to pixel centers. Three important functions that satisfy this requirement are p ROT c, that rotates point p by 90 degrees counterclockwise around the point c, and p I_FLIP i_0, p J_FLIP j_0, that mirror the point p around the lines $i = i_0$ and $j = j_0$, respectively, where c, i_0, and j_0 have integer or half-integer coordinates.

Another operation that falls in this category is the *sampling* operation, defined by the identity $(\mathsf{P}\ \mathsf{SAMPLE}\ [s, t])_{[is, jt]} \equiv \mathsf{P}_{[i,j]}$ (s and t must be integers). Its effect is to collect every sth row and tth column of P and "compress" the resulting picture.

A related operation is b REPEAT F, which denotes the bitmap obtained by replicating a rectangular patch of the picture F all over the image plane. The box b defines the portion of F to be replicated, and also determines the phase (placement of the first "tile") and the period of tesselation.

2.4. Some Fundamental Pictures

In this section we define some fundamental pictures that either possess wide applicability or are relevant to some major application of raster-scan graphics. These pictures are the "atoms" of our bitmap calculus.

The notation ALL v stands for a picture whose pixels are all equal to the bitstring v. The expressions I_IM n *and* J_IM n represent a signed picture of depth n where pixel $[i, j]$ is the value of i or j, respectively, extended or truncated to n bits. These two primitives allow the description of pictures where pixel $[i, j]$ is defined by an expression involving i and j. Note, however, that all pictures described in this way will be ultimately periodic, owing to the finite number of bits of i and j used according to the definition.

An important class of pictures is that of *geometric shapes,* which in the bitmap calculus are defined as discrete approximations to the geometric figures of the

Euclidean plane geometry, like rectangles, triangles, circles, and so forth. Such images are defined as masks which are TRUE inside the figure and FALSE outside it.

A box can be considered a geometric shape and used as such in formulas. Additional geometric shapes one would like to have as primitives very much depend on the application, so their definition is better left to the user of the calculus. Two primitives that seem to be reasonably useful in graphics applications, however, are CIRCLE[c, r] (a circle centered at point c with radius r) and TRIANGLE[p_1, p_2, p_3] (a triangle with the specified vertices). Also useful is LINE[p_1, p_2, w], a line segment from p_1 to p_2 with thickness w. However, the exact definition of this shape, particularly at the two endpoints, is again application-dependent.

3. THE MUMBLE LANGUAGE

MUMBLE is a language designed to perform computations on digital images or bitmaps. It implements the operations and primitives discussed in the previous section, so formulas of the bitmap calculus are generally valid MUMBLE expressions. As most programming languages, this one includes many features that have no counterpart in the language of mathematics, the most notable being the concepts of *assignment* and *variable* (in the sense of assignable identifier).

In previous sections we used the term "bitmap" both as a loose synonym for "discrete image," and in the machine-related sense meaning "an area of the memory where a discrete image is stored." The distinction between the two concepts is quite important for understanding the semantics of MUMBLE; therefore, from now on we reserve the term "bitmap" for the machine-related concept only, and use the terms "image" or "picture" for the abstract concept.

Pictures are always infinite in both dimensions, whereas bitmaps are necessarily restricted to a finite subset of the picture plane, which in the case of MUMBLE is a finite *bounding box*. It follows that when a picture is stored in a bitmap, only the portion that falls inside its bounding box will be preserved. If the content of the bitmap is to be used in subsequent computations, it is extended to an infinite picture by assuming the pixels outside the box to be zero.

Thanks to these conventions, it has been possible for us to give a "natural" meaning to arbitrarily complex expressions, involving geometric figures and transformations, bitmaps of different sizes, and infinitely repeating patterns, without having to specify for *each* operation what happens at the points where some operand is undefined.

We should also emphasize here that the definition of pixels as bitstrings, rather than as pure integers or real numbers, is an essential part of the bitmap calculus we propose, not a limitation imposed on MUMBLE by implementation issues. This assumption lends a different character to the picture computations we are concentrating on from those traditionally associated with image processing, where pixels are considered to be real-valued.

MUMBLE is a block-structured, weakly typed interactive language. Besides bitstrings (which include numbers), bitmaps, and pictures, it handles a few other kinds of objects, such as characters and character strings, pointers, records, variables, and procedures that are necessary in a useful programming environment.

3.1. Values and Variables; Bitmaps

In general, a MUMBLE expression may evaluate to either a *value* or a *variable.* A value is a piece of data, or bit pattern (some examples are bitstrings, pictures, character strings, and pointers to other objects); a variable is an object in which values may be stored. *Ordinary variables* are similar to the simple variables of PASCAL or ALGOL, except that they are essentially typeless: they can be assigned any simple value, including bitstrings and pointers. *Pixel variables* can only be assigned bitstring values of a fixed size and type; moreover, they cannot be named or created in isolation, occurring only as elements of bitmaps.

A bitmap B behaves in MUMBLE as a rectangular array of pixel variables; subscripting can be used to access each pixel individually. Ordinary variables can be directly named in MUMBLE, but bitmaps and individual pixel variables cannot. The usual way to refer to a bitmap in MUMBLE is to store its *descriptor* (essentially a pointer to its location in memory, plus some associated information) in an ordinary variable. To get hold of the bitmap itself, the descriptor must be dereferenced by the "!" operator.

The *ordinary assignment* V := *e* stores the value of the expression *e* into the ordinary variable V, without any type conversions. Pixel variables cannot be assigned with ":=": they require the *pixel assignment* operator "←", as for example in (!B)[i, j] ← (!B)[i, j] + 1. Unlike ":=", the "←" operator extends or truncates the value on the right-hand side to conform to the variable on the left. The pixel assignment operator can also be used to assign a whole picture to an entire bitmap. The command (!B) ← (!B) + ALL 1 will add one to every pixel of the bitmap pointed to by B.

Actually, MUMBLE automatically dereferences a bitmap descriptor when an indexing or pixel assignment operation is applied to it, so we could have written B[i, j] ← B[i, j] + 1 and B ← B + ALL 1 in the examples above. Note that B := A assigns to B the *descriptor* in A, while B ← A copies the *contents* of the bitmap pointed by A into the one pointed by B. The expression BOUNDS A returns the bounding box of bitmap A.

Applying a geometric transformation (SHIFT, ROT, etc.) to a bitmap descriptor returns another descriptor, pointing to the *same* bitmap, with its parameters modified so as to simulate the specified transformation. That is, the pixel assignment (A SHIFT [−2, 3])[i, j] ← 1 has exactly the same effect as A[i + 2, j − 3] ← 1.

The meaning of an assignment like A := A SHIFT [−2, 3] should be carefully considered: the picture stored in the bitmap A stays fixed, but the *names* of its pixels and its bounding box are moved around. It is like sliding an egg carton (the bitmap) over a tiled floor (the image plane): the positions of the eggs (pixels) change, even though they do not move relative to the carton.

In contrast, the assignment A ← A SHIFT [−2, 3] will actually move the contents of the bitmap by the specified amount, without disturbing its descriptor or bounding box; some pixel values around the margins will be lost, and some will be filled in with zeros. Continuing the above analogy, this corresponds to leaving the carton in the same place and moving each egg inside it by the specified amount.

To determine the meaning of pixel assignments that refer to the same bitmap both on the left and on the right-hand side, as in the above example, we must take into account the order in which the pixels are assigned by MUMBLE. The right-hand side is evaluated, and the left-hand side is written, in the so-called *ravel order*: this is a double loop, the inner loop being from low j to high j, and the outer loop being from low i to high i. Of course, the new value of a pixel has to be computed before being written; but, apart from this, owing to internal buffering in the implementation, no implicit guarantee is made about the relative time ordering of the reads versus the writes.

The effect of A ← A SHIFT [2, 3] is therefore undefined, since the value of $A_{[i-2,j-3]}$ may already have been overwritten by the time the individual pixel assignment $A_{[i,j]} \leftarrow A_{[i-2,j-3]}$ is executed. Note however that the assignments A ← NOT A and A ← (A SHIFT [0, − 1]) are perfectly safe. Only in some cases will dangerous assignments like the above be flagged by the interpreter.

It should be noted that the bitblt instruction discussed in Section 1 usually chooses the order of evaluation in such a way that no pixel is overwritten before being read; this increases significantly both the conceptual simplicity and the usefulness of the bitblt instruction. It would be highly desirable to include the same mechanism in the bitmap assignment of MUMBLE; unfortunately, the generality of the expressions allowed in such assignments makes this technique much harder to implement. Indeed, some bitmap assignments cannot be correctly evaluated on a pixel-by-pixel basis, no matter what the scanning order is. Consider, for example, the statement A ← A I_FLIP 5, where the bounding box of A is [[0, 0], [10, 10]]: no matter which of the two assignments $A_{[0,0]} \leftarrow A_{[9,0]}$ and $A_{[9,0]} \leftarrow A_{[0,0]}$ is executed first, the other will produce the wrong result.

This problem can be solved by the introduction of auxiliary bitmaps (e.g., by writing Aux ← A I_FLIP 5; A ← Aux in the example above). Considering, however, that bitmaps are typically quite large objects, and that the detection of such invalid assignments is by no means trivial, we decided against the automatic introduction of auxiliary bitmaps by MUMBLE, leaving to the user the primary responsibility of detecting and preventing such errors. Section 4.5 shows how one can code a common case of the bitblt instruction in the MUMBLE language.

3.2. Why Bitmap Descriptors?

Bitmap descriptors and the existence of two flavors of assignment operator are among the main causes of complexity in the MUMBLE language. These two features were motivated by the desire to combine a natural way of expressing bitmap computations with enough structure to make a smart evaluator possible. For example, consider the following computation:

```
A ← B OR (((B SHIFT [0, 1]     + (B SHIFT [0, −1])
                + (B SHIFT [1, 0]) + (B SHIFT [−1, 0])) > ALL 2)
```

An important feature of MUMBLE is that complex picture expressions like the one above do not require temporary bitmaps for intermediate results. Since bitmaps are usually quite large objects, this is a feature of considerable value.

However, this optimization is usually inhibited by the common programming practice of breaking complex expressions into a sequence of smaller ones and assigning their values to temporary variables. If we rewrite the statement above as

```
BL ← B SHIFT [0, 1]; BR ← B SHIFT [0, −1];
BU ← B SHIFT [1, 0]; BD ← B SHIFT [−1, 0];
A ← B OR ((BL + BR + BU + BD) > ALL 2);
```

the computation will require the four temporary bitmaps BL, BR, BU, and BD, each containing a slightly displaced copy of the bitmap B. If we write instead

```
BL := B SHIFT [0, 1]; BR := B SHIFT [0, −1];
BU := B SHIFT [1, 0]; BD := B SHIFT [−1, 0];
A ← B OR ((BL + BR + BU + BD) > ALL 2);
```

then BL, BR, BU, and BD will just be modified *descriptors* of the original bitmap B.

3.3 Deferred Evaluation and Picture Formulas

An important aspect of MUMBLE is the mechanism by which such apparently infinite expressions like B + ALL 1 are evaluated in finite time. This is accomplished by evaluating picture expressions and subexpressions only when and where their values are actually needed. For example, in A ← B AND (C OR D), the expression B AND (C OR D) will not be computed outside the bounding box of the bitmap A; furthermore, if $B_{[i,j]}$ is zero, it is possible that MUMBLE will skip the evaluation of $(C \text{ OR } D)_{[i,j]}$.

A conditional picture expression like IF A THEN B ← F ELSE C ← F FI, where all variables evaluate to bitmap descriptors, means that the assignment $B_{[i,j]} \leftarrow F_{[i,j]}$ is to be performed for all pixels, and only those, for which $A_{[i,j]}$ is not FALSE. For the remaining pixels, and only those, the assignment $C_{[i,j]} \leftarrow F_{[i,j]}$ is to be performed. A conditional expression including bitmap assignments can be replaced by an equivalent one containing Boolean operations instead of the conditionals, but in that form its execution cost may be much higher.

The operations in a picture expression like A ← B AND (C OR D) are not evaluated in the "bottom-up" fashion characteristic of PASCAL or ALGOL. Rather, the whole expression is first encoded into a *delayed picture formula*, or *form*, that is a symbolic, unevaluated representation of those operations and their operands. The actual evaluation proceeds in a "top-down" fashion: the outermost operation ("←" in the example) is examined first, and its operands are evaluated only for the pixels where their values are actually needed. Inner operations are treated in the same way. This process is called the *scanning* of the picture formula, and its details will be discussed in Section 5.

It is possible to build and keep around a deferred formula in its "unscanned" state: the (ordinary) assignment F := FORM {B AND (C OR D)} will store in F a pointer to the given deferred formula, without actually carrying out the indicated picture computations. The formula F can be used to build more complex expressions and statements, as for example A ← (X AND F) OR (Y AND NOT F). Complex picture expressions can thus be built up little by little, but no actual evaluation happens until the programmer indicates that this should occur.

A deferred formula may contain bitmap assignment operations, as in the case of E := FORM {B ← B + ALL 1}. It is frequently the case that a formula like E has to be evaluated only for its side effects, and its value (if any) is to be discarded. This is accomplished in MUMBLE by the statement SCAN E.

Deferred formulas are similar to picture-returning procedures in many ways, with F := FORM{. . .} corresponding to a procedure declaration, and A ← F or SCAN F corresponding to its invocation. The two concepts differ in several aspects, however, the most notable being the way in which their bodies are evaluated.

3.4. Other Syntactical Features

This section presents an overview of some unconventional features of MUMBLE syntax. Together with the previous sections, the paragraphs below should enable a reader with programming experience to follow the examples in Section 4. For a full presentation of the language, see [5].

Syntactically, a MUMBLE expression consists of closed subexpressions (identifiers, unary operators, literals, parenthesized expressions), *applied* to each other and/or combined by means of *infix operators*. For example, in

```
B i + Foo [i, j] XOR (Func R) Fum X
```

the identifier B is applied to i, Foo is applied to [i, j], and the result of applying Func to R is applied to the result of Fum applied to X; the symbols "+" and "XOR" are infix operators. The priorities of infix operators are for the most part similar to those of ALGOL. Most usual operation on numbers, strings, etc. are available in MUMBLE.

The application construct has different meanings depending on the types of the operands. It is used to denote procedure invocation, unary operator application, bitmap, record, and string subscripting (with the subscript on the right, as in X i) and record field selection (with the selector on the left, as in im X). Applications are evaluated from right to left, so Fee Fi Foo Fum means Fee (Fi (Foo Fum)).

A single MUMBLE construct implements both the record concept of PASCAL and the block structure of ALGOL. The MUMBLE expression

$$[L_0 : E_0, L_1 : E_1, L_2 : E_2, \ldots, L_{n-1} : E_{n-1}]$$

constructs an n-field record, with selectors $L_0, L_1, \ldots, L_{n-1}$ and whose values are the results of evaluating $E_0, E_1, \ldots, E_{n-1}$ (some or all L_i and/or E_i may be omitted). During the evaluation of E_i, the field selectors L_0 to L_{n-1} can be used as if they were ordinary variables. A pointer to the constructed record is returned as the value of the whole expression. The infix operator ";", that evaluates both its arguments in sequence and returns the second, can be used with ordinary parentheses to build compound statements.

Individual components of a record can also be accessed by subscripting, and can be modified by the ordinary assignment operator ":=". Record fields are untyped: they may hold any simple value. Record constructors can be nested, and the usual ALGOL-like scope rules apply.

Points and vectors of the bitmap calculus are represented in MUMBLE by records with two bitstring components, I and J. Only the numerical values of I and J, and not their lengths and types, are relevant for MUMBLE operations that require point and vector arguments.

The language contains the usual collection of conditional and iterative commands, including IF–FI, DO–OD, WHILE, FOR, EXIT, and so forth. Procedures are declared by writing *name*: PROC {*formal parameters* | *local declarations and commands*}. If the type of a formal parameter is not specified, any simple value can be passed as argument. The result can be any simple value, and is given by the last expression appearing in the procedure body, or by the execution of RETURN *e*.

4. EXAMPLES

Some of the MUMBLE syntax that was not explained in Section 3 is elaborated in the comments to the programs below. Comments are set off by the string "—" and are terminated either by "—" or by a carriage return.

4.1. Exchange of Bitmaps within Box

One of the many fun uses of XOR is the one below, which exchanges the contents of the two bitmaps A and B within box bx.

```
Exchange: PROC {A: BITMAP,
                B: BITMAP,
                bx: BOX |
                  Aw: A CUT bx,   —descriptor for portion of A within bx.
                  Bw: B CUT bx,   —descriptor for portion of B within bx.
                  Aw ← Aw XOR Bw;
                  Bw ← Aw XOR Bw;
                  Aw ← Aw XOR Bw
                }
```

Note that no temporary bitmaps are used. For more tricks like this see [10].

4.2. Area-Filling by XORing Scan Lines

In this scheme we assume that we are given a simple rookwise connected[2] closed pixel path in the plane which nowhere touches itself. The path is specified as a bitmap with 1's where the pixels on the path are, and 0's elsewhere. The algorithm below fills in (makes 1's) all pixels interior to the path, by essentially replacing each scan line with the XOR of all scan lines up to (and including) that one. To avoid some boundary problems, the rightmost 1 of each run of 1's in a scan line is removed before this operation is applied. At the end, the path is ORed back in to restore the rightmost pixels of runs that might have been missed.

```
Fill: PROC
  {Bd : BITMAP |   — assume Bd is a bitmap with bounds [[0, 0], [N, N]], i.e. an N × N
                     array.
  N: J HIGH BOUNDS Bd,   — extract the value of N, given Bd.
  CN: [[0, 0], [1, N]] BITMAP OF DEPTH 1,   — a one line temporary bitmap,
                                            — initially aligned with line 0 of Bd.
```

[2] This means that successive pixels on the path are obtained from each other by moving one up, down, left, or right.

```
  C:,  — a temporary variable.
  CN ← ALL 0;
  FOR L IN [0. . N) DO  — for each scan line (L = 0, 1, . . . , N − 1) do:
    C:= Bd CUT [[L, 0], [L + 1, N]];  — get descriptor for line L of Bd.
    CN ← CN XOR (C AND (C SHIFT [0, −1]));  — remove rightmost 1 in each run
                                            — of 1's in scan line L and XOR
                                            . it into CN.
    C ← CN OR C;  — write CN back into Bd, adding original path.
    CN := CN SHIFT [1, 0]  — align CN with the next scan line
  OD
}
```

The condition that the path not touch itself is essential. Fig. 2 shows a path for which the notion of interior is ambiguous.

4.3. Rotation of a Bitmap by Shearing

We mentioned in Section 1.2 that it is possible to rotate an $n \times n$ bitmap with roughly $3n$ bitblts. By rotation here we mean the actual rearrangement of bitmap pixels in memory, not just a descriptor transformation. The simplest way to get this effect in MUMBLE is by executing the assignment T ← A ROT c; A ← T, where c is the center of the bounding box of A. However, the technique used in the following example is instructive, as similar methods can be used for bitmap transposition (for example, see [2]).

Fig. 3 illustrates how the technique works on a small example.

```
SRotate: PROC
  {M: BITMAP, N: INT |  — assume the bounds of M are [[0, 0], [N, N]]
    A: [[0, 0], [2*N − 1, 2*N − 1]] BITMAP OF DEPTH 1;
    — circularly rotate rows of M right by I (Figure 3b).
    FOR I IN [0. .N − 1] DO
      A CUT [[I, 0], [I + 1, 2*N − 1]] ← M CUT [[I, 0], [I + 1, N]] SHIFT [0, I])
    OD;
    M ← A CUT [[0, 0], [N, N]] + A CUT [[0, N], [N, 2*N − 1]];
    — circularly rotate columns of M down by N − 1 − J (Figure 3c).
    FOR J IN [0. .N − 1] DO
      A CUT [[0, J], [2*N − 1, J + 1]] ← M CUT [[0, J], [N, I + 1]] SHIFT [N − J −
      1, 0]
    OD;
    M ← A CUT [[0, 0], [N, N]] + A CUT [[N, 0], [2*N − 1, N]];
    — circularly rotate rows of M left by N − 1 − I (Figure 3d).
    FOR I IN [0. .N − 1] DO
      A CUT [[I, 0], [I + 1, 2*N − 1]] ← M CUT [[I, 0], [I + 1, N]] SHIFT [0, I + 1]
    OD;
    M ← A CUT [[0, 0], [N, N]] + A CUT [[0, N], [N, 2*N − 1]]
  }
```

4.4. Rotation by the Use of Masks

The following rotation algorithm uses a "binary mask" technique, based on the work of Floyd [10, 2]. The bitmap size is assumed to be $2^n \times 2^n$, and the rotation is accomplished in n steps. After k steps, the current picture is the original one except that it has been divided into square subarrays of size $2^k \times 2^k$, and each

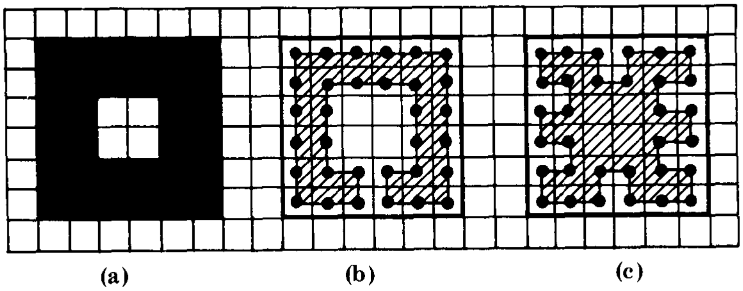

Fig. 2. Example of a picture (a) which can be interpreted as a simple rookwise connected path in two different ways (b), (c).

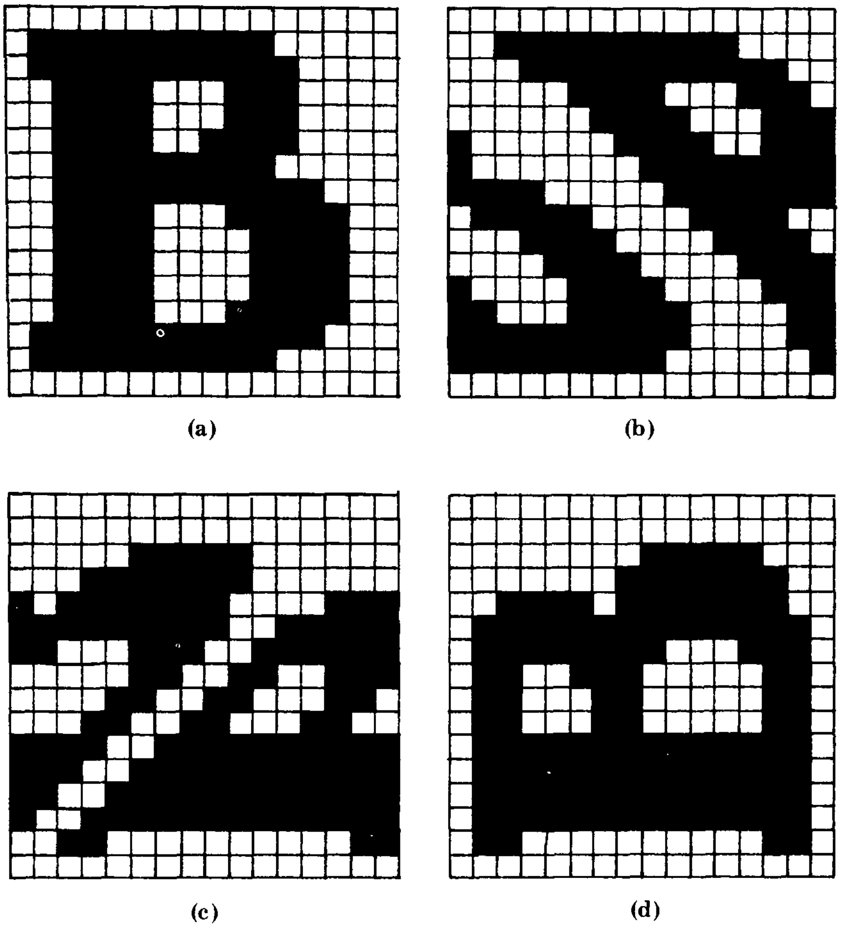

Fig. 3. Bitmap rotation by shearing. (a) is the original bitmap, (b) shows it after shearing horizontally, (c) shows it after shearing vertically, and (d) shows it after shearing horizontally again.

subarray has been rotated in place. The next step will assemble each four of these subarrays into a single rotated subarray of size $2^{k+1} \times 2^{k+1}$. All groups are operated upon simultaneously; the subarrays to be moved in each group are picked out by a mask containing squares of ones of size $2^k \times 2^k$ in the appropriate places.

```
MRotate: PROC
 {n: INT, B: BITMAP |
   T: [[0, 0], [2↑n, 2↑n]] BITMAP OF DEPTH (DEPTH B);
   FOR k IN [0. .n) DO   — for k = 0, 1, . . . , n − 1 do:
     m: 2↑ k;            — current subarray size
     — Maskij selects subarray [i, j] in each group of four
     Mask11: FORM{(I_IM n LAYER k) AND (J_IM n LAYER k)};
     Mask00: Mask11 SHIFT [−m, −m];
     Mask01: Mask11 SHIFT [−m, 0];
     Mask10: Mask11 SHIFT [0, −m];
   T ← (Mask00 AND B) SHIFT [m, 0] OR
       (Mask01 AND B) SHIFT [0, −m] OR
       (Mask11 AND B) SHIFT [−m, 0] OR
       (Mask10 AND B) SHIFT [0, m];
   B ← T
  OD
}
```

Fig. 4 illustrates the process. The advantage of this method is that the bitmap is always scanned in the natural order, and therefore the computation is quite fast. This method is especially well suited for bitmaps whose size is of the order of the machine word size.

4.5. Simulation of the Bitblt Instruction

As we mentioned in Section 3, the typical bitblt instruction chooses the order of individual pixel assignments in such a way that no pixel is overwritten before being read. Although this feature is not automatically provided by the bitmap assignment of MUMBLE, it can be easily programmed in most simple cases, including in particular those where a bitblt instruction would be used.

The procedure below simulates the bitblt instruction D ← D OR S, where D and S are two rectangular subarrays of a given bitmap B. The two operands of the bitblt are specified by their bounding boxes Dbox and Sbox, which may overlap. The bitblt is simulated by a MUMBLE bitmap assignment, with the descriptor of S appropriately shifted so that its upper left corner is aligned with that of D. The pixel evaluation order is effectively modified by mirroring (the descriptors of) both S and D around either or both coordinate axes, depending on the relative placement of the two boxes.

```
ORbitblt: PROC
 {B: BITMAP, Dbox: BOX, Sbox: BOX |
   D: B CUT Dbox,             — descriptor for destination
   S: B CUT Sbox,             — descriptor for source
   v: LOW Sbox − LOW Dbox,    — relative displacement between S and D
   IF I v < 0 THEN
     D := D I_FLIP 0;    — mirror v and the descriptors
     S := S I_FLIP 0     — of S and D around i-axis
```

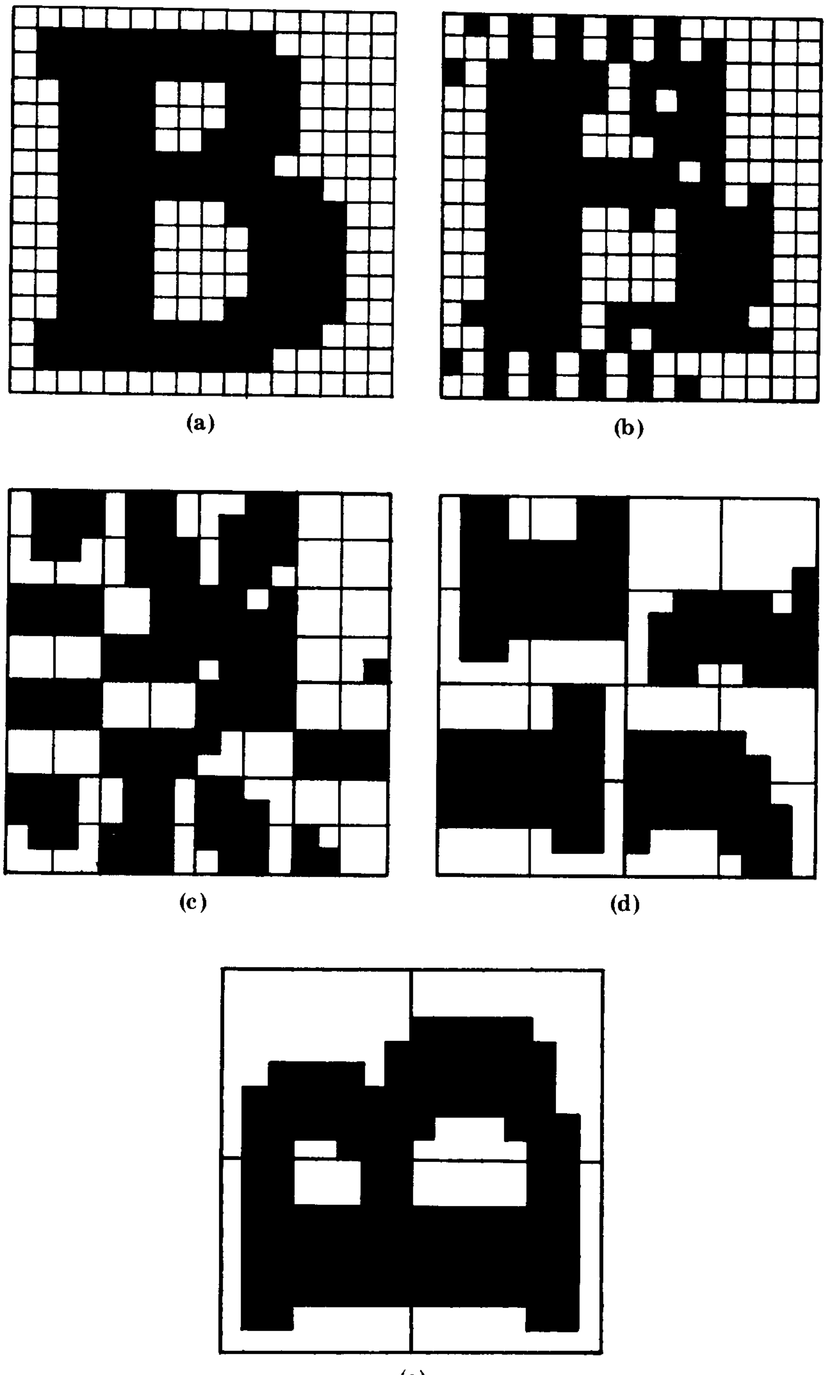

Fig. 4. Bitmap rotation by Floyd's algorithm. (a) is the original bitmap, with $n = 4$; (b)–(e) show the result of each iteration ($k = 0$ through 3).

```
    FI;
    IF J v < 0 THEN
        D := D J_FLIP 0;   — mirror v and the descriptors
        S := S J_FLIP 0    — of S and D around j-axis
    FI;
    —Perform the assignment:
    D ← D OR S
}
```

The procedure above can be extended in a straightforward way to implement other features of typical bitblt instructions, like the provision for other Boolean operations besides OR, and the ability to use an infinitely repeated stipple pattern instead of the source S.

5. THE IMPLEMENTATION OF BITMAP COMPUTATIONS

BOP (an acronym for Bitmap Operation Package) is a collection of procedures that implements the construction and modification of bitmaps and deferred formulas, and the scanning of the latter. BOP includes also some routines for the manipulation of boxes, points, and vectors. The current implementation of BOP is written in MESA [14] and runs on the Dorado [11] personal computers developed at Xerox PARC. However, the techniques used by BOP are fairly general and can be implemented in a wide range of languages and machines.

After discussing briefly the internal representation of bitmaps and formulas, we present three optimization techniques that BOP uses in the scanning of picture formulas: deferred evaluation, run encoding, and word-oriented processing.

Similar ideas have appeared earlier in the literature in the context of other languages dealing with "large" objects. See, for instance, [7] for a discussion of lazy evaluation in LISP, [6] for delayed evaluation in APL, and [4] for related notions in SNOBOL. As with other very high-level languages, the MUMBLE implementation is neither a pure interpreter, nor a pure compiler. The current execution style is certainly closer to an interpreter, but as more is learned about where the bottlenecks of the system are, we may shift to an earlier binding of some of the operators, and thus closer to compilation.

5.1. Bitmaps and Bitmap Descriptors

Bitmaps are internally represented by descriptors which contain the following parameters:

TYPE (signed or unsigned) and DEPTH;
Bounding box (BOUNDS);
Addressing increments (ISTEP and JSTEP);
Address of pixel [0, 0] (BASE).

The addressing increments are integer parameters used in computing the locations of individual pixels. The address of the leading bit of pixel $[i, j]$ is given by BASE + $i\cdot$ISTEP + $j\cdot$JSTEP.

The geometric transformations of bitmaps discussed in Section 3.3 are implemented very efficiently in BOP, since they require only that the parameters

BOUNDS, BASE, ISTEP, and JSTEP be appropriately modified. When a bitmap is created, the parameter JSTEP is equal to the pixel length, and ISTEP is the total line length (JSTEP times the number of pixels per line). These relations will generally cease to hold if the bitmap is transformed.

A bitmap with JSTEP = ±DEPTH is said to be *compact*. Compact bitmaps, and especially those with unit JSTEP, can be handled much more efficiently than general ones.

5.2. Picture Formula Representation

A picture formula is represented by an acyclic data structure whose nodes are called picture instructions. The leaves of this data structure correspond to "atomic" picture formulas: geometric forms, bitmaps, uniform bit fields, and so on. Internal nodes correspond to the operators used to build the picture formula; for example, after the assignments below are evaluated

```
F1 := FORM{TRIANGLE[P1, P2, P3] OR CIRCLE[C1, R1]};
F2 := FORM{F1 XOR (F1 + ALL 1)};
F3 := FORM{IF CIRCLE[C2, R2] THEN B ← F1 ELSE C ← F2 FI};
```

the internal representation of the FORMs stored in F1, F2 and F3 will be that given by Figure 5.

Bitmap descriptors are incorporated in deferred formulas as parameters of *bitmap reader* and *bitmap writer* nodes. The evaluation of a bitmap reader (which has no subformulas) results in the current contents of the bitmap. The region outside the bounding box will return zeros.

A bitmap writer node corresponds to a picture assignment operation like B ← F; it has a single subformula (F) and its parameter record contains a descriptor of the receiving bitmap (B). Besides such "pure" bitmap writers, BOP provides also for *bitmap modifiers*, which have the same structure but perform the computation B ← B ⊙ F, where ⊙ is a bitwise Boolean operation. (There is no special syntax in MUMBLE for creating such nodes, but a moderately smart interpreter can easily detect many simple cases in which they are applicable.) As it turns out, performing a simple assignment $b \leftarrow f$ to a bit field b of a machine word is as expensive as any operation of the form $b \leftarrow b \odot f$, and frequently more so. Thus the decision to have bitmap modifiers as a special kind of node is mainly one of efficiency.

5.3. Evaluation of Picture Formulas

As we mentioned before BOP evaluates (scans) a picture formula in a top-down fashion: the root of the formula is examined first, and evaluation proceeds to its subformulas only if and when their values are needed. In this process, each subformula is evaluated for some set of indices $[i, j]$, which is enumerated in increasing lexicographical order. This sequence of pixel positions is broken by BOP into a number of *segments* of varied length; the segments are processed sequentially, and the pixels in each segment are evaluated in parallel (each operation in the formula is applied simultaneously to the whole segment).

We can view the evaluation of a subformula S for one segment as being initiated by a "request," coming from its parent node, that specifies the position p of the

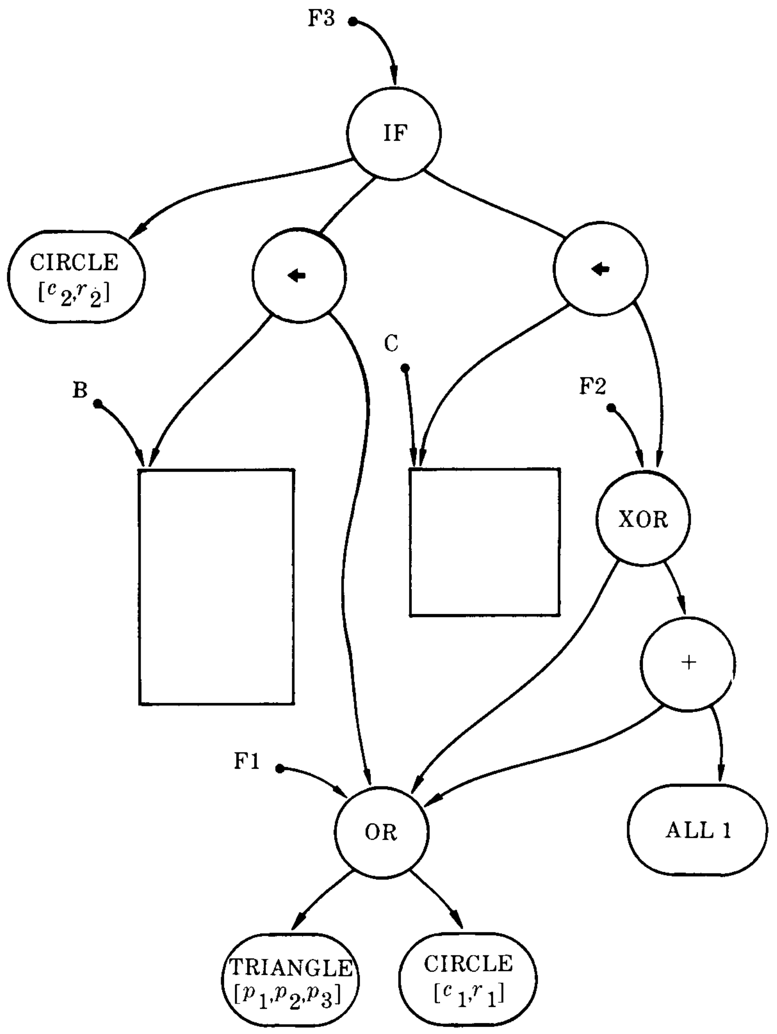

Fig. 5. The internal representation of a deferred picture formula.

next pixel of S needed by the parent. To satisfy the request, S eventually returns a segment of its value, beginning at pixel p, in its result buffer. The subformula is free to choose the length of the segment, as long as it is nonzero, and must inform the parent of its choice. Subsequent requests will obtain their value from the buffer, until it is exhausted. Of course, in order to satisfy a request, S may have to pass additional requests to its own subformulas, and so on.

Geometric primitives (circles, triangles, etc.) check the position of the requested pixel against the boundary of the figure, and return a string of 1's or 0's, as appropriate. Bitmap readers and writers check the indices of the requested pixel against the bounding box of the bitmap; if it is outside, the request is trivially satisfied. Otherwise, a bitmap reader will extract a few more pixels from the bitmap and make them into a picture segment; a bitmap writer will send a request to the formula on the right side of the "$\leftarrow$", and will store the returned segment into the bitmap.

Although this top-down scheme is more complex than the usual bottom-up one, it saves a great deal of unnecessary computation. For example, in the scanning of B $\leftarrow$ F, the formula F will be automatically evaluated only for the pixels which fall inside the bounding box of the bitmap B. Also, if X returns a segment consisting of all 1's during the scanning of X OR Y, BOP will avoid the evaluation of the corresponding segment of Y.

The evaluation by segments used by BOP is intermediate between the two extremes of (1) evaluating the formula sequentially at each pixel in turn (a pure top-down strategy), and (2) applying each operation in turn to all pixels of its operands (a pure bottom-up strategy). Although scheme (2) saves time by traversing the formula only once, it requires one full-size bitmap for each intermediate result in the formula. In scheme (1), the temporary variables have to hold a single pixel each, but the formula is traversed and interpreted once for each pixel.

In BOP's scheme the temporaries need only be large enough to hold a single segment of a picture; as we shall see below, the size of the segments is adjusted so that each fits in a few words of storage. The time overhead due to repeated formula traversals is reduced over that of scheme (1) by a factor equal to the average number of pixels in a segment. In view of the fixed costs incurred by all three schemes, it can be seen that it does not pay to increase the size of the temporaries beyond a certain point. With 10 pixels per segment we already get 90 percent of the time savings that could be obtained using full-sized temporary bitmaps.

Unfortunately, the bitblt instructions available in most computers can only be used according to scheme (2) and require full-sized intermediate bitmaps even for moderately complex expressions. A smarter implementation of MUMBLE and/or BOP might be capable of detecting simple cases (like B $\leftarrow$ B OR C) where bitblt instructions could be used.

5.4. Segment Representation and Run Encoding

BOP represents a segment of a picture of depth k by the indices of its first and last pixels and k binary words $W_{k-1}, W_{k-2}, \ldots, W_0$ containing the pixel values. The pixels are represented in the so-called *vertical format*: W_0 contains the least significant bit of all pixels, W_1 contains the next one, and so on. The word W_i is therefore a segment of the ith layer of the picture.

This rather unconventional representation simplifies considerably the lamination, truncation, and padding of pictures, and the extraction of picture layers, and it allows the uniform handling of pictures of arbitrary depth, independently of the machine word size w. This representation is also ideal for exploiting the full power of word-oriented machine instructions. Consider for example the scanning of the picture expression A $\leftarrow$ B + C, where A has depth 3 and B, C have depth 2. BOP evaluates this expression for w pixels simultaneously using only six full-word Boolean operations, without using expensive field extractions.

Of course, conversion to and from the "horizontal" format is required when pictures are to be stored into or retrieved from bitmaps. The vertical format also becomes less and less efficient as the pixel size approaches the machine's word size w; at this point, the individual processing of each pixel becomes a reasonable alternative.

The size of a picture segment, as specified by its initial and final indices, may be more than w pixels; BOP assumes in this case that all pixels beyond the first w are identical to the wth, the last one that is actually represented in the W words. In this way, arbitrarily long runs of pixels with uniform values can be encoded by a single segment. It turns out that this run encoding method combines nicely with the vertical format. All point operations can be carried out on the first w pixels of the segment, and the "replicate the last pixel" convention will automatically produce the correct result for the rest.

This technique is particularly important when evaluating geometric primitives (circles, triangles, and so forth), since they can be represented by a small number of uniform segments. It is less valuable for pictures obtained from bitmaps, as the main cost here comes from extracting the pixels and converting them to vertical format; however, the area outside the bitmap bounds, being uniformly zero, will produce at most two segments per scan line.

5.5. Miscellaneous Optimizations

When two pictures are combined in a binary operation like A OP B, it is usually the case that the segments produced by A and B do not match in size and position. BOP will further subdivide those segments until they match. This may cause excessive fragmentation of the pictures, which could be corrected by condensing short consecutive segments of the result into longer ones. (The current implementation of BOP does not perform this optimization, however.)

The FORM construct of MUMBLE encourages the use of shared subformulas, like F1 in Figure 5. When an expression is scanned, a shared subformula (one that can be reached following more than one path from the root) may receive two or more requests for the same pixels. By remembering the last segment returned in a private buffer, the subformula can satisfy repeated requests with a single evaluation, sometimes with considerable savings in execution time.

A geometric transformation (shift, rotation, or the like) applied to a deferred formula creates a copy of the corresponding data structure, with its parameters modified so as to produce the transformed picture when scanned. Usually, the modifications affect only the leaves of the structure (bitmap descriptors and geometric primitives), since, for example, (A OR B) SHIFT [2, 3] is equivalent to (A SHIFT [2, 3]) OR (B SHIFT [2, 3]). For similar optimizations in the case of APL, see [6].

BOP guarantees that, during the scanning of an expression, successive evaluation requests reaching a subformula refer to nondecreasing pixel indices; as a consequence, those pixels will be evaluated in a mostly sequential manner. Therefore, incremental techniques become both simple and effective in the addressing of bitmap elements and in the computation of the boundaries of geometric figures.

A complete implementation of MUMBLE and BOP requires an efficient storage management system with automatic garbage collection. The current implementation relies on the facilities provided by the Cedar programming environment under development at Xerox PARC.

6. CONCLUSIONS

In this paper we have presented a bitmap calculus and a set of ideas for its efficient implementation. It may be instructive to draw a comparison between BOP and other raster graphics systems, such as those based on the bitblt instruction discussed in Section 1.2. The main difference concerns the size of the context over which one is optimizing. A software package based on bitblt alone will do very well in an environment where most raster manipulations occur in isolation and correspond to simple rectangle movements, as with a single bitblt. This is certainly the case in applications like text-editors, where extensive graphic manipulations are not required. BOP, on the other hand, looks at a wider context and is aimed at optimizing situations where a lot of bitmap computations are being done, one after the other. The sharing and deferred evaluation techniques incorporated in BOP will yield significant advantages in that case. Thus BOP may be the right implementation in an environment where graphic raster manipulations are the main activity of the system. We hope to gain enough experience with our MUMBLE/BOP system to validate this claim.

The other contribution of this paper has to do with the promotion of the bitmap calculus. We have collected several interesting bitmap algorithms besides those shown in Section 4, including the game of life [3], self-replicating automata [22], Levialdi's transform [12], scan-conversion by diffusion [8], picture cleanup [19], and fractal curves [13]; and we are certain that many similar algorithms remain to be discovered. It is fascinating to us how local and uniform computations can be used to obtain globally significant results. This, of course, also suggests that one day an efficient implementation of MUMBLE may be possible using special-purpose VLSI chips.

ACKNOWLEDGMENTS

The authors wish to thank Dan Kolkowitz, who was partly responsible for the design and implementation of an earlier version of BOP, and Lyle Ramshaw for his careful reading of the manuscript. Thanks are also due to the referees of this paper and the TOG editors, whose valuable and detailed comments immensely aided the exposition.

REFERENCES

1. BECHTOLSHEIM, A. AND BASKETT, F. High-performance raster graphics for microcomputer systems. *Computer Gr. 14,* 3 (July 1980), 43–47.
2. FLOYD, R. W. Permuting information in idealized two-level storage. In *Symposium on the Complexity of Computer Computations,* Plenum Press, New York, 1972, pp. 105–110.
3. GARDNER, M. The game of life. *Scientific American,* Oct. 1970, Feb. 1971, and Jan. 1972.
4. GRISWOLD, R. E. AND GRISWOLD, M. T. *A SNOBOL 4 Primer,* Prentice-Hall, Englewood Cliffs, N. J., 1973.

5. Guibas, L. J. and Stolfi, J. MUMBLE Reference Manual, in preparation.
6. Guibas, L. J. and Wyatt, D. K. Compilation and delayed evaluation in APL. In *Fifth Annual Symposium on Principles of Programming Languages* ACM, New York, 1978, pp. 1–8.
7. Henderson, P. and Morris, H. J., Jr. A lazy evaluator. In *Third Annual Symposium on Principles of Programming Languages*, ACM, New York, 1976, pp. 95–103.
8. Ingalls, D. Scan-Conversion by Diffusion. Personal communication, 1976.
9. Iverson, K. E. *A Programming Language.* Wiley, New York, 1962.
10. Knuth, D. E. *The Art of Computer Programming.* Combinatorial Algorithms, vol. IV. (Preprint version) Stanford University, Stanford, Calif. 1977.
11. Lampson, B. W., et al. The Dorado: A high performance personal computer. Xerox PARC Technical Report CSL 81-1, January 1981.
12. Levialdi, S. On shrinking binary picture patterns. *Commun. ACM 15,* 1 (Jan. 1972), 7–10.
13. Mandelbrot, B. *Fractals*: *Form, Chance and Dimension.* W. Freeman, San Francisco, Calif., 1977.
14. Mitchell, J., et al. Mesa language manual—version 5.0. Xerox PARC Technical Report CSL 79-3, April 1979.
15. Newman, W. M. and Sproull, R. F. *Principles of Interactive Computer Graphics.* Second edition, McGraw-Hill, New York, 1979.
16. Pakin, S. *APL\360 Reference Manual.* Science Research Associates, 1970.
17. Introduction to Raster Graphics, ACM Siggraph '81 Conference Tutorial, 1981.
18. Thacker, C. P., et al. Alto—A personal computer. Xerox PARC Technical Report CSL 79-11, August 1979.
19. Van Wyk, C., and Knuth, D. E. A programming and problem solving seminar. Technical Report STAN-CS-79-707, Stanford University, Stanford, Calif., 1979.
20. Von Neuman. J. *Theory of Self-Replicating Automata.* Univ. of Illinois (Urbana) Press, 1966.
21. Weinreb, D. and Moon, D. Lisp Machine Manual. M.I.T., 1981.
22. Winograd, T. Self-replicating automata. Unpublished note, 1967. See also *Scientific American*, Feb. 1971.

Received March 1982; revised May 1982; accepted May 1982.